A Formally Proved, Complete Algorithm for Path Resolution with Symbolic Links

Ran Chen, Martin Clochard, Claude Marché

A Formally Proved, Complete Algorithm for Path Resolution with Symbolic Links

Ran Chen
Institute of Software, Chinese Academy of Science, Beijing, China
Martin Clochard
LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
and
Claude Marché
Inria, Université Paris-Saclay, F-91120 Palaiseau

In the context of file systems like those of Unix, *path resolution* is the operation that given a character string denoting an access path, determines the target object (a file, a directory, etc.) designated by this path. This operation is not trivial because of the presence of *symbolic links*. Indeed, the presence of such links may induce infinite loops in the resolution process.

We consider a path resolution algorithm that always terminates, detecting if it enters an infinite loop and reports a resolution failure in such a case. We propose a formal specification of path resolution and we formally prove that our algorithm terminates on any input, and is correct and complete with respect to our formal specification.

1. INTRODUCTION

The problem of *path resolution* takes place in the context of the *file system* component of operating systems. It is the operation that, given a *pathname*, determines the target object (typically a file or a directory) it denotes in the current file system, if any. In particular for the operating systems of the Unix family, target objects can also be *symbolic links*: objects that themselves denote a pathname. When meeting a symbolic link, path resolution must proceed with resolution of the pathname denoted by that link. The presence of symbolic links gives to the path resolution process a recursive nature, that may lead to non-termination if caution is not taken.

The goal of this paper is to consider an algorithm for path resolution that carefully takes care of the potentially non-terminating situations, and formally prove this algorithm correct with respect to a formal specification expressing the expected functional behavior of path resolution.

In this paper, we adopt the notations of Unix file systems, as they are standardized by the Portable Operating System Interface (POSIX) IEEE family of standards [IEE]. A pathname is a character string that is made of a sequence of *filenames* separated by the special character `"/"`. A pathname is *absolute* if it starts with `"/"` and *relative* otherwise. A filename, also called *pathname component* in POSIX, is a non-empty sequence of characters, containing neither `"/"` nor

Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) and by the CoLiS project (ANR-15-CE25-0001, https://www.irif.fr/~treinen/colis/) of the French national research organization

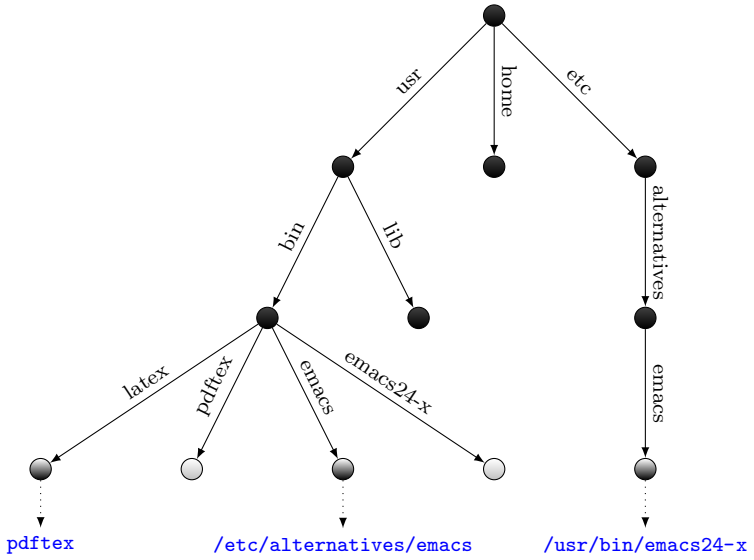

Fig. 1. Excerpt of the file system tree typically found in a Debian installation. Black nodes denote directories, white nodes denote regular files, and gray nodes denote symbolic links.

the NUL character (i.e. ASCII code 0). The filenames `"."` and `".."` have special meanings, respectively the current and the parent directory. When the given pathname is absolute, pathname resolution starts from the root directory, otherwise it starts from the *current* directory of the process that attempts resolution. Figure 1 presents an excerpt of the real file system tree that appears in a computer with a typical Debian installation. Notice the relative symbolic link `/usr/bin/latex` that points to `pdftex`, that is `/usr/bin/pdftex` as an absolute path, and the absolute symbolic link `/usr/bin/emacs` that points to `/etc/alternatives/emacs` which is itself a symbolic link to `/usr/bin/emacs24-x`.

In practice, they are several possible causes for a path resolution failure. For example, a pathname may denote an existing object but resolution can fail if the user has insufficient access permissions. Our goal is to focus on the difficulty induced by the presence of symbolic links, hence we are going to abstract away other aspects such as permissions. The difficulty with symbolic links is that the file system tree becomes some kind of a graph in which symbolic links may define cycles. A simple example would be a symbolic link that points to itself, or, on Figure 1, if the link `/etc/alternatives/emacs` was pointing to `/usr/bin/emacs` instead of `/usr/bin/emacs24-x`. In the presence of such cycles, the pathname resolution algorithm must be careful not to go into an infinite loop. This is the issue we address in this paper. This issue is solved in practice by setting an arbitrary bound on the number of symbolic links that can be traversed during a given path resolution[1]. It means that the typical algorithm for path resolution implemented in a real OS is an incomplete one. To our knowledge, the question of the existence of

[1]POSIX requires this number to be at least 8 (`http://pubs.opengroup.org/onlinepubs/9699919799/`, constant `_POSIX_SYMLOOP_MAX`)

a terminating and complete algorithm for this problem has never been investigated in an academic point of view. The only source of information we found where this question was discussed is on a few discussion threads on the Web[2]. Indeed, it is not hard to invent such an algorithm from scratch, by adapting known ideas from classical graph traversal algorithms. The algorithm we use in this paper is designed like that. The hard question we address is not how to design such an algorithm, but how to formally prove it correct. Our main inspiration, for designing both the formal specification and the formal proof, comes from formally verified graph-traversal algorithms. However, the issue with symbolic links is significantly different from graph-traversal specification and proofs we have seen in the literature. The hardest task was to discover an appropriate invariant preserved across the recursive calls of the algorithm, sufficient to prove the completeness of resolution: when the algorithm reports a failure then it is true that no valid resolution exist.

Concretely, the formalization is done using the Why3 program verifier [FP13]. This system permits to write algorithms that include non-purely functional features such as in-place modification and exceptions. Algorithms can be given formal specifications, under the form of annotations such as pre- and postconditions. The Why3 system generates verification conditions from annotated programs, that can be discharged by several possible automated provers. Bobot et al. [BFMP15] present an introduction to the use of Why3 on some case studies. Many examples can be found in the Why3 gallery of verified programs[3].

In Section 2 we first present how we model file systems. We then present our resolution algorithm in Section 3. We describe our formal specification in Section 4 and show how the algorithm is proved in Section 5. Section 6 presents some conclusions and related work. The complete code for this work, annotated with formal specifications, is available at URL `http://toccata.lri.fr/gallery/path_resolution.en.html`. For simplicity, in this paper we only consider the case of absolute links, the case of relative links being very similar. We considered both cases in the complete code, which is also presented in a research report [CCM16] giving the full details of the proof results.

2. ABSTRACT MODEL OF THE FILE SYSTEM

For our development, we formalize the file system in an abstract way. Regular files play no role in the symbolic link issues of path resolution algorithm so we just ignore them. The file system is seen as a directed graph where the vertices are directories, called *dirnodes*. Edges of this graph are labeled by file names. An edge from a dirnode d_1 to a dirnode d_2 labeled by f means that f is a name that belongs to directory d_1 and that points to the sub-directory d_2.

2.1 Pathnames

A *pathname* is a sequence of file names, separated by slash characters, used to identify a file or a directory in the file system. In a pathname, "." and ".." have a special meaning, respectively to denote the current directory and the parent directory. We formalize them abstractly as follows.

[2] e.g. `http://unix.stackexchange.com/questions/99159/is-there-an-algorithm-to-decide-if-a-symlink-loops`
[3] `http://toccata.lri.fr/gallery`

```
type dirnode
constant root : dirnode
type child =
  | Absent
  | Dir dirnode
  | AbsLink path
function lookup dirnode filename : child
function parent dirnode : dirnode
  axiom parent_root: parent root = root
  axiom parent_non_root: forall d1 f d2. lookup d1 f = Dir d2 → parent d2 = d1
```

Fig. 2. Formalization of the file system as a graph

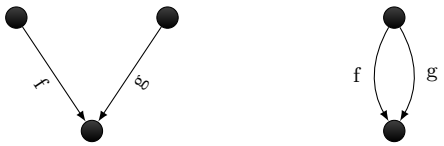

Fig. 3. Parent of a dirnode must be unique. Shape at left is forbidden whereas shape at right is allowed.

```
type filename
type pathcomponent = Down filename | Up | Here
type path = list pathcomponent
```

The type for filenames is left abstract, since for our purpose we don't need to know anything about it. A path is a list of path components, which can be either `Up` to denote `".."`, `Here` to denote `"."`, or (`Down` f) to denote a normal filename f.

2.2 The file system

The graph formed by the file system is formalized using the declarations given on Figure 2. The constant `root` denotes the root directory of the file system. The function `lookup` is a total function which looks up a filename of a directory and returns the corresponding *child*. A *child* could be of three kinds, namely `Absent` denoting that this filename does not appear in that directory, (`Dir` d) denoting that it exists and points to a sub-directory d, and (`AbsLink` p) meaning it is a symbolic link that stores an absolute path p. We finally declare the function `parent` to get the parent directory of some dirnode. We axiomatize that function with two axioms. The first axiom `parent_root` indicates that the parent directory of root is root itself. The second axiom `parent_non_root` specifies that if we can lookup a filename f from directory d_1 to directory d_2, then d_1 is the parent directory of d_2.

It should be noted that the introduction of the parent function implies that the underlying graph is almost a tree: it is not possible to have two different dirnodes pointing to the same sub-directory, as in the left part of Figure 3. Yet, the shape of the right part of the same figure is allowed. Notice that this second situation is forbidden to occur in a Unix filesystem (at least for directories), but we don't need to rule it out since our algorithm still works in such a case. Notice finally that in this formalization, nothing requires the graph to be finite.

Example 1. Considering the structure of Figure 1, we have

```
lookup root "usr" = Dir d1
lookup d1 "bin" = Dir d2
lookup d2 "emacs" = AbsLink "/etc/alternatives/emacs"
lookup d2 "foo" = Absent
```

3. RESOLUTION ALGORITHMS

We can give a naive path resolution algorithm now. We start to resolve a path p from some directory d. We match the path with several cases.

—If it's an empty path, then we stay in directory d.
—If the path starts with "`..`" , then we go to the parent directory of d and resolve the rest of the path.
—If it starts with "`.`", then we stay in the current directory d and resolve the rest of the path.
—If it starts with a normal file name, then we lookup the filename in directory d:
  —If it is absent, then path p resolves to nowhere. We raise an error then.
  —If it is a directory d'. then we resolve the remaining path from d'.
  —If it denotes an absolute link ps, then we recursively resolve the path ps from root to some directory d', and then resolve the remaining path of p from d'.

Here is the corresponding Why3 code for this naive algorithm.

```
exception Error

let rec aux_resolve (d:dirnode) (p:path) : dirnode =
  match p with
  | Nil → d
  | Cons Up pr → aux_resolve (parent d) pr
  | Cons Here pr → aux_resolve d pr
  | Cons (Down f) pr →
      match lookup d f with
      | Absent → raise Error
      | Dir d' → aux_resolve d' pr
      | AbsLink ps →
        let d' = aux_resolve root ps in
        aux_resolve d' pr
      end
  end
```

The naive algorithm above doesn't check the existence of loops in the path. Because of the presence of symbolic links, we may have a loop in the path, thus the path cannot be resolved to anywhere and the algorithm keeps looping forever. Figure 4 presents some examples of such loops. `/a/e` is a path with a loop in it since there is a symbolic link that points to itself. `/c/f` is also a path with a loop in it because there are two symbolic links in the path and they point to each other. From these two examples, one may suggest that we can detect a loop in the path by recording the symbolic links we meet in it, and stop if we traverse a symbolic link for a second time. Unfortunately such a check would be wrong because it is too much restricted, as shown by the example `/b/d/d/d/d/c` in which we meet the same symbolic link d several times. But this path can be resolved successfully.

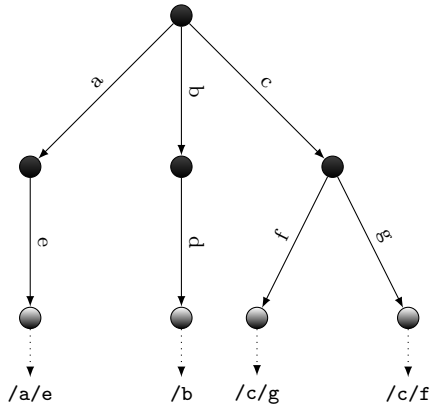

Fig. 4. Toy examples of partial path resolution.

From the third example above, we can now present a better algorithm to resolve a path. It has an extra parameter `active` needed to detect loops: it is a set of pairs made of a directory node and a file name that resolves to a symbolic link at that directory. Each time we meet a symbolic link (d, f) in the path, we check if it is already present in the active set, so as to detect the repetition of symbolic links when we resolve the link itself. Here is the Why3 code of the algorithm.

```
1  type lnk = (dirnode,filename)
2
3  let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
4  = match p with
5    | Nil → d
6    | Cons Up pr → let d' = parent d in aux_resolve d' pr active
7    | Cons Here pr → aux_resolve d pr active
8    | Cons (Down f) pr →
9      match lookup d f with
10     | Absent → raise Error
11     | Dir d' → aux_resolve d' pr active
12     | AbsLink ps →
13       if mem (d,f) active
14       then raise Error
15       else begin
16         let actadd = add (d,f) active in
17         let d' = aux_resolve root ps actadd in
18         aux_resolve d' pr active
19       end
20     end
21   end
```

The resolving function `aux_resolve` keeps looking up the path components recursively. Every time we meet a symbolic link in the path, we store the link in the active set used for resolving the link itself (line 17). If we meet the same link again, we know that the link is looping and the path could not be resolved. On the other hand, notice in the second recursive call (line 18) that the active set does not contain the link anymore, that is after resolution of a link is done, resolution continues with the rest of the path, which is allowed to traverse the link again.

4. FORMAL SPECIFICATION OF PATH RESOLUTION

Our goal is now to express in a formal way the informal property "from some directory d_1 we can resolve a path p to some other directory d_2". Because resolution does not always succeed, we cannot formalize this property as a total function (that from d_1 and p would return d_2) in a classical two-valued logic like those of Why3, in which all functions are total. Instead, we formalize this property as a ternary predicate, that we denote in this paper as $d_1, p \rightsquigarrow d_2$.

We define this predicate *inductively*, that is we define it as the smallest predicate satisfying the rules below. The notation $d(f)$ is an abbreviation for `lookup` $d\ f$.

$$\frac{}{d, \varepsilon \rightsquigarrow d} \quad \text{(ResolveNil)}$$

$$\frac{d_1(f) = \texttt{Dir}\ d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, f/p \rightsquigarrow d_3} \quad \text{(ResolveDir)}$$

$$\frac{d_1(f) = \texttt{AbsLink}\ ps \quad root, ps \rightsquigarrow d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, f/p \rightsquigarrow d_3} \quad \text{(ResolveAbsLink)}$$

$$\frac{\texttt{parent}\ d_1 = d_2 \quad d_2, p \rightsquigarrow d_3}{d_1, ../p \rightsquigarrow d_3} \quad \text{(ResolveUp)}$$

$$\frac{d_1, p \rightsquigarrow d_2}{d_1, ./p \rightsquigarrow d_2} \quad \text{(ResolveHere)}$$

The first rule means that resolving the empty path from some node d results to d itself. The second rule means that if from node d_1 the filename f denotes a directory node d_2, and if from d_2 the path p resolves to some node d_3, then we know that from node d_1 we can resolve the path f/p to node d_3. The third rule indicates that if from some node d_1 we look up filename f and meet an absolute link which stores a path ps, if we resolve this link from root to some node d_2 and if from d_2 we can resolve path p to d_3, then from node d_1 we can resolve path f/p to node d_3. The fourth rule means if d_2 is the parent directory of some node d_1, and from d_2 we can resolve path p to some node d_3, then the path $../p$ can be resolved from d_1 to d_3. The last rule handles similarly the case of a path $./p$.

The predicate $d_1, p \rightsquigarrow d_2$ can be formalized in the Why3 logic using an inductive definition, as follows, that corresponds closely to the rules above.

```
inductive resolve_to dirnode path dirnode =
  | ResolveNil : forall d. resolve_to d Nil d
  | ResolveDir : forall d1 f d2 p d3.
    lookup d1 f = Dir d2 → resolve_to d2 p d3 →
    resolve_to d1 (Cons (Down f) p) d3
  | ResolveAbsLink : forall d1 f ps p d2 d3.
    lookup d1 f = AbsLink ps → resolve_to root ps d2 → resolve_to d2 p d3 →
    resolve_to d1 (Cons (Down f) p) d3
  | ResolveUp: forall d1 d2 d3 p.
    parent d1 = d2 → resolve_to d2 p d3 → resolve_to d1 (Cons Up p) d3
  | ResolveHere: forall d1 p d2.
    resolve_to d1 p d2 → resolve_to d1 (Cons Here p) d2
```

Example 2. Here are some examples of valid resolution, in the file system of Figure 1. First we pretend that resolving the path `/usr/bin` from root results in

d_2, that is $root, \texttt{usr/bin} \rightsquigarrow d_2$. The proof of this fact is

$$\dfrac{root(\texttt{usr}) = d_1 \quad \dfrac{d_1(\texttt{bin}) = d_2 \quad \overline{d_2, \varepsilon \rightsquigarrow d_2}}{d_1, \texttt{bin} \rightsquigarrow d_2}}{root, \texttt{usr/bin} \rightsquigarrow d_2}$$

Suppose the parent directory of some directory node d_1 is `root`, and we pretend that resolving the path `../etc/alternatives/emacs` from d_1 results in d_4. The proof of $d_1, \texttt{../etc/alternatives/emacs} \rightsquigarrow d_4$ is

$$\dfrac{\texttt{parent } d_1 = \texttt{root} \quad \dfrac{\texttt{root(etc)} = d_2 \quad \dfrac{d_2(\texttt{alternatives}) = d_3 \quad \Pi_1}{d_2, \texttt{alternatives/emacs} \rightsquigarrow d_4}}{\texttt{root, etc/alternatives/emacs} \rightsquigarrow d_4}}{d_1, \texttt{../etc/alternatives/emacs} \rightsquigarrow d_4}$$

where Π_1 is the proof

$$\dfrac{d_3(\texttt{emacs}) = \texttt{AbsLink(/usr/bin/emacs24-x)} \quad \Pi_2 \quad \overline{d_4, \varepsilon \rightsquigarrow d_4}}{d_3, \texttt{emacs} \rightsquigarrow d_4}$$

and Π_2 is the proof

$$\dfrac{\texttt{root(usr)} = d_5 \quad \dfrac{d_5(\texttt{bin}) = d_6 \quad \dfrac{d_6(\texttt{emacs24-x}) = d_4 \quad \overline{d_4, \varepsilon \rightsquigarrow d_4}}{d_6, \texttt{emacs24-x} \rightsquigarrow d_4}}{d_5, \texttt{bin/emacs24-x} \rightsquigarrow d_4}}{\texttt{root, usr/bin/emacs24-x} \rightsquigarrow d_4}$$

4.1 Comparison with POSIX specification of resolution

If we compare our formal specification of path resolution with the informal one of POSIX[4], we can notice a slight divergence, lying on the way symbolic links must be handled: "If a symbolic link is encountered during pathname resolution, [...] the system shall prefix the remaining pathname, if any, with the contents of the symbolic link". In other words, in our rule ResolveAbsLink, we should not have two premises but only one to resolve the concatenation of the link and the remaining pathname.

Our definition is indeed simpler because it does not use concatenation, and in particular it will make the proofs easier. To show that there is no difference with POSIX informal specification, we now define another predicate closer to POSIX specification and prove the equivalence between this specification and the one above. The predicate $d_1, p \underset{\text{POSIX}}{\rightsquigarrow} d_2$ is defined inductively with the same rules as $d_1, p \rightsquigarrow d_2$ except the rule for symbolic links which becomes as follows, where the operator $+\!\!+$ denotes the concatenation of paths.

$$\dfrac{d_1(f) = \texttt{AbsLink } ps \quad root, ps +\!\!+ p \underset{\text{POSIX}}{\rightsquigarrow} d_2}{d_1, f/p \underset{\text{POSIX}}{\rightsquigarrow} d_2} \quad \text{(ResolveAbsLinkPOSIX)}$$

[4] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_13

We want to prove that the predicate above is equivalent to the first one, as stated by the following theorem.

THEOREM 3. *For any directory node d_1, d_2, and any path p,*

$$d_1, p \underset{POSIX}{\rightsquigarrow} d_2 \qquad \textit{if and only if} \qquad d_1, p \rightsquigarrow d_2$$

To prove this theorem we have to state a few auxiliary lemmas. The first lemmas below are related to the first resolution predicate.

LEMMA 4. *for all dirnodes d_1, d_2, d_3 and paths p, q, if $d_1, p \rightsquigarrow d_2$ and $d_2, q \rightsquigarrow d_3$ then $d_1, p +\!\!+ q \rightsquigarrow d_3$.*

The proof is done by induction on the hypothesis $d_1, p_1 \rightsquigarrow d_2$. Within Why3 such a lemma can be stated directly as follows.

```
lemma resolve_to_append : forall d1 d2 d3 p q.
  resolve_to d1 p d2 → resolve_to d2 q d3 → resolve_to d1 (p ++ q) d3
```

The proof is done using the Why3 transformation `induction_pr`, that applies an induction scheme corresponding to the inductive definition of the resolution predicate. The resulting goals are then proved by automatic provers. From now on we do not mention anymore the Why3 code of our lemmas, and we will come back to proof results in practice in Section 5.4. See also the report [CCM16] for the full Why3 code and the detailed proof results.

We establish the converse property of Lemma 4 using two other lemmas. We denote by operator :: the list cons.

LEMMA 5. *for all dirnodes d_1, d_3, any path component c and any path p, if $d_1, c :: p \rightsquigarrow d_3$ then there exists d_2 such that $d_1, c :: \mathtt{Nil} \rightsquigarrow d_2$ and $d_2, p_2 \rightsquigarrow d_3$.*

This lemma is proved by induction on hypothesis $d_1, c :: p \rightsquigarrow d_3$.

LEMMA 6. *for all paths p_1, p_2 and all dirnodes d_1, d_3, if $d_1, p_1 +\!\!+ p_2 \rightsquigarrow d_3$ then there exists d_2 such that $d_1, p_1 \rightsquigarrow d_2$ and $d_2, p_2 \rightsquigarrow d_3$.*

This lemma is proved by structural induction on p_1 seen as a list, and using the previous lemma.

The next lemma concerns the POSIX variant of resolution predicate. It is similar to Lemma 4.

LEMMA 7. *For any directory node d_1, d_2 and d_3, and any path p_1, p_2, if $d_1, p_1 \underset{POSIX}{\rightsquigarrow} d_2$ and $d_2, p_2 \underset{POSIX}{\rightsquigarrow} d_3$ then $d_1, p_1 +\!\!+ p_2 \underset{POSIX}{\rightsquigarrow} d_3$*

The proof is done by induction on hypothesis $d_1, p_1 \underset{POSIX}{\rightsquigarrow} d_2$.

Finally, the proof of Theorem 3 is done by considering each direction of the equivalence separately, and reasoning by induction on the predicate in hypothesis in both cases.

4.2 Resolution Indexed with Explicit Height

For the purpose of proving the completeness of our resolution algorithm, we need to explicitly refer to the height of the proof of some judgment $d_1, p \rightsquigarrow d_2$. More precisely, we will derive that some resolution cannot exist by proving that its height

would satisfy contradictory inequalities. We thus introduce another predicate with an extra argument corresponding to that height. Moreover, in order to express that some path p can *not* be resolved, we say that it can be resolved with an *infinite* height. We denote $d_1, p \twoheadrightarrow o$ to mean "the outcome of resolving path p from node d_1 is o". The outcome o is either a pair (d_2, h) meaning "it resolves to node d_2 with a proof of height h", or $o = \infty$ meaning that the resolution does not succeed. That predicate is also defined inductively with the rules below.

$$\frac{\forall d_1.\ \neg(d, p \rightsquigarrow d_1)}{d, p \twoheadrightarrow \infty} \quad \text{(ResolveHeightAbsent)}$$

$$\frac{}{d, \varepsilon \twoheadrightarrow (d, 0)} \quad \text{(ResolveHeightNil)}$$

$$\frac{d_1(f) = \texttt{Dir}\ d_2 \quad d_2, p \twoheadrightarrow (d_3, h)}{d_1, \texttt{f/p} \twoheadrightarrow (d_3, h+1)} \quad \text{(ResolveHeightDir)}$$

$$\frac{d_1(f) = \texttt{AbsLink}\ ps \quad \texttt{root}, ps \twoheadrightarrow (d_2, h_1) \quad d_2, p \twoheadrightarrow (d_3, h_2)}{d_1, \texttt{f/p} \twoheadrightarrow (d_3, \max(h_1, h_2) + 1)} \quad \text{(ResolveHeightAbsLink)}$$

$$\frac{\texttt{parent}\ d_1 = d_2 \quad d_2, p \twoheadrightarrow (d_3, h)}{d_1, \texttt{../p} \twoheadrightarrow (d_3, h+1)} \quad \text{(ResolveHeightUp)}$$

$$\frac{d_1, p \twoheadrightarrow (d_2, h)}{d_1, \texttt{./p} \twoheadrightarrow (d_2, h+1)} \quad \text{(ResolveHeightHere)}$$

We need some technical lemmas about resolution with height. First, we need to state that proof height of a resolved path is greater than or equal to 0. Of course this is a trivial property for human beings, but our formalization in Why3 uses mathematical integers for heights, so this has to be stated and proved.

LEMMA 8. *For any directory node d_1 and d_2, and any path p, and any height h if $d_1, p \twoheadrightarrow d_2, h$ then $h \geq 0$*

The proof can be done by induction on the hypothesis $d_1, p \rightsquigarrow d_2$, and looking at all the cases of rules applied to establish $d_1, p \rightsquigarrow d_2$.

We also need lemmas that relate the resolution predicate with explicit height to the original resolution predicate. First, if there is a resolution with explicit height $d_1, p \twoheadrightarrow (d_2, h)$ with any finite height h, then there is a resolution $d_1, p \rightsquigarrow d_2$.

LEMMA 9. *For any directory nodes d_1 and d_2, any path p, and any height h if $d_1, p \twoheadrightarrow d_2, h$ then $d_1, p \rightsquigarrow d_2$.*

The proof can be done by induction on the hypothesis $d_1, p \rightsquigarrow d_2$, and looking at all the cases of rules applied to establish $d_1, p \rightsquigarrow d_2$.

A second lemma works the other way around: every time we resolve a path to some directory we can always resolve with some height.

LEMMA 10. *For any directory node d_1 and d_2, and any path p, if $d_1, p \rightsquigarrow d_2$ then there exists h such that $d_1, p \twoheadrightarrow (d_2, h)$.*

The proof proceeds also by induction.

The last set of lemmas concern the determinism of resolution: the target node of path resolution is unique, if it exists. The following lemma states this property.

LEMMA 11. *For any directory node d_1, d_2 and d_3, and any path p, if $d_1, p \rightsquigarrow d_2$ and $d_1, p \rightsquigarrow d_3$ then $d_2 = d_3$*

A similar property exists for the predicate on resolution with height, stating that two resolutions of the same path must have the same outcome (finite or infinite).

LEMMA 12. *For any directory node d, and any path p, and any outcomes o_1 and o_2, if $d, p \not\rightsquigarrow o_1$ and $d, p \not\rightsquigarrow o_2$ then $o_1 = o_2$*

The last lemma below is saying that we can always find an outcome (possibly infinite) for any path resolving from any directory.

LEMMA 13. *For any directory node d, and any path p, there exists o such that $d, p \not\rightsquigarrow o$*

5. PROOF OF THE PATH RESOLUTION ALGORITHM

5.1 Termination

Up to now, we did not specify that the filesystem has finitely many nodes. Potentially, resolution could not terminate even if there is no loop: imagine a link l_1 pointing to another link l_2 itself pointing to l_3 etc.

To prove termination, we thus need to add more constraints in our model of the filesystem, as follows.

```
use import set.FSet              (* finite sets, from Why3's standard library *)
constant alllinks : set lnk      (* a finite set *)
axiom alllinks_in : forall d f ps.
  lookup d f = AbsLink ps → mem (d,f) alllinks
```

In other words, there exists some finite set `alllinks` of pairs (dirnode,filename) such that all links in the file system belong to `alllinks`. Indeed, we show that our algorithm always terminates even if the file system is infinite: only the number of symbolic links must be assumed finite.

To achieve the proof of termination, we just need to state a *variant*, that is a quantity that decreases at each recursive call. A proper variant in this case is as follows: either the active set increases, or it remains unchanged and the length of the path p decreases. Instead of saying that the `active` set increases, we say that the complement set (`alllinks` − `active`) decreases. This requires to add a precondition stating that the `active` set is always a subset of `alllinks`. Such a precondition acts as an invariant maintained for all the recursive calls.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  requires { subset active alllinks }
  variant { cardinal alllinks - cardinal active, p }
= ...
```

From the variant above, given as a pair of an integer and a list, Why3 implicitly considers that the associated well-founded ordering is the lexicographic composition of the natural ordering on non-negative integers and the sub-list ordering on lists. The proof of termination is then obtained by automatic provers.

5.2 Correctness

The correctness of the algorithm is stated using the following post-condition.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  ensures { resolve_to d p result }
```

The proof is very easy, because the recursive calls of the algorithm recursively construct the needed premises to build the inductive proof of $d, p \rightsquigarrow \texttt{result}$. Though, notice that for this proof it is important to use our first variant of the resolution predicate, and not the POSIX one.

5.3 Completeness

The completeness is stated using the following post-condition stated when the function raises the exception `Error`.

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  raises { Error → forall d'. not resolve_to d p d' }
```

The hardest part of our formal proof is to prove this completeness property. We need to add more invariants on the active set, again under the form of preconditions to be satisfied by the recursive calls.

The invariants on the `active` set are as follows: for all $(d_1, f) \in \texttt{active}$ and for any ps, if $d_1(f) = \texttt{AbsLink}\ ps$ then

—$\forall d_2.\ (\texttt{root}, ps \rightsquigarrow d_2) \rightarrow \exists d'.\ (d, p \rightsquigarrow d')$
—$\forall d_2, h_1, d', h.\ (\texttt{root}, ps \not\rightsquigarrow d_2, h_1) \rightarrow (d, p \not\rightsquigarrow d', h) \rightarrow h \leq h_1$

Both assertions say something about an arbitrary pair (d_1, f) in the active set. If the filename f in dirnode d_1 denotes an absolute link to some path ps, then:

—The first assertion states that if ps is resolvable from root (to some d_2) then p is resolvable from d (to some d').
—The second assertion states that if ps is resolvable from root with some finite height h_1 and if p is resolvable from d with some finite height h then h is smaller than h_1.

In other words, these two assertions together mean that if you resolve any pair in the active set, then the input path p is resolvable also, and the proof must have a smaller height. This is the key property that allows us to prevent cycles in the proofs of path resolution, as we will see below. Still another way to express this is to say that the resolution of the current path p is a part of the resolution of all the paths that appear in the active set.

The code is thus annotated as shown on Figure 5. The assertion in the first line of the body of `aux_resolve` is added to guide the automatic provers to instantiate Lemma 13. Notice the use of an option type to encode outcomes of resolution with height.

The proof of the exceptional post-condition must be done in the case of each occurrence of `raise Error` in the code. The first case, when the considered filename does not exists (line 20), is easy: no rules for constructing a proof of resolution can apply. The other case (line 24) concerns the symbolic links. By contradiction, if we assume that it is possible to resolve d, p to some d', then this proof has some finite height h. But then since (d, f) is in the active set and points to `AbsLink` ps, the first part of the invariant says that $\texttt{root}, ps$ is resolvable. Moreover, the second part of the invariant says that the height of that resolution is some $h_1 \geq h$. By

```
let rec aux_resolve (d: dirnode) (p:path) (active:set lnk) : dirnode
  requires { subset active alllinks }
  requires { forall d1 f ps d2.
               mem (d1, f) active → lookup d1 f = AbsLink ps →
               resolve_to root ps d2 → exists r. resolve_to d p r }
  requires { forall d1 f ps d2 h1 d' h.
               mem (d1,f) active → lookup d1 f = AbsLink ps →
               resolve_with_height root ps (Some(d2, h1)) →
               resolve_with_height d p (Some(d', h)) → h ≤ h1 }
  ensures { resolve_to d p result }
  raises { Error → forall d'. not resolve_to d p d' }
  variant { cardinal alllinks - cardinal active, p }
= assert { exists h. resolve_with_height d p h }; (* to help provers *)
  match p with
  | Nil → d
  | Cons Up pr → let d' = parent d in aux_resolve d' pr active
  | Cons Here pr → aux_resolve d pr active
  | Cons (Down f) pr →
    match lookup d f with
    | Absent → raise Error
    | Dir d' → aux_resolve d' pr active
    | AbsLink ps →
      if mem (d,f) active
      then raise Error
      else begin
        let actadd = add (d,f) active in
        let d' = aux_resolve root ps actadd in
        aux_resolve d' pr active
      end
    end
  end

let resolve (d: dirnode) (p:path) : dirnode
  ensures { resolve_to d p result }
  raises { Error → forall d2. not resolve_to d p d2 }
= aux_resolve d p empty
```

Fig. 5. Code annotated with formal specifications

applying the rule ResolveAbsLink we can then build a proof of resolution of d, p of height $h' = 1 + \min(h_1, h'')$ where h'' is the height of the proof of resolution of the remaining path `pr`. Hence $h' \geq h_1 + 1$, but by uniqueness of resolution h' must be equal to h, contradicting $h_1 \geq h$.

5.4 Proof results

The table of Figure 6 summarizes the provers' results on all the verification conditions of our development, including preliminary lemmas. The total number of VCs is 198. We run all provers on all VCs with a time limit of 10 seconds. The first column gives the number of VCs successfully proved by the given prover. The other columns give respectively the minimum, average and maximum time the prover took to solve the VCs it proved. The last column gives the number of VCs that are proved only by the given prover. This number is 0 for Z3, meaning that Z3 is not

Prover	number of VCs solved	min time	max time	average time	number of VCs solved only by this prover
Coq (8.5pl3)	1	0.52	0.52	0.52	1
CVC3 (2.4.1)	115	0.01	2.89	0.20	9
CVC4 (1.4)	136	0.01	2.37	0.14	8
Alt-Ergo (1.01)	119	0.00	8.17	0.41	7
Eprover (1.8-001)	125	0.01	7.87	0.26	6
Z3 (4.4.1)	108	0.00	6.25	0.26	0

Fig. 6. Summary of proof results

really needed, but all the other provers are needed to make a complete proof of our development.

Notice that we needed one Coq proof to solve one VC in the part where we prove the equivalence between our definition of resolution and the one closer to POSIX informal definition. This Coq proof is not complex at all (only 9 lines, see [CCM16]) and does not involve any complex reasoning step (e.g. such as induction). It requires however to provide explicit instantiations for quantified formulas, which is probably the reason why it is not discharged by our automatic provers. See [CCM16] for the details of each verification condition and which transformations and provers we used to discharge them.

6. CONCLUSIONS

We designed a formal specification of the intended meaning of pathname resolution in a file system involving symbolic links. We considered an algorithm that is not limited in the number of traversed symbolic links, and we formally proved that this algorithm is terminating, correct and complete. The most difficult part of this work is to design an adequate definition of the meaning of path resolution under the form of a ternary predicate $d_1, p \rightsquigarrow d_2$, and also, in order to achieve the formal verification of the algorithm, to discover an appropriate invariant, expressed using an extended variant of this predicate, indexed with an explicit height of the derivation. Our approach was designed so that the proofs can be performed with automatic provers, with the exception of a single, yet simple, proof that was done in Coq.

This idea of using the height of the derivation is a new lesson we learned during this work. In particular, such a concept have not be used so far in the formal verification of other algorithms for graph traversal. Yet, reasoning on proof trees is routinely used in formalized reasoning about semantics of programming languages, in particular when using a proof assistant like Coq where proofs are first-class objects. Even when structural recursion is not enough, reasoning on proof size or proof height is of course possible. However, in a context like in Why3 where proofs are not first-class objects, and where proofs are mainly done using automatic provers, it is new to us that such an inductive reasoning on proofs can be performed, by adding an extra parameter (like the height for us) to the considered inductive predicate. Although, adding an explicit parameter to an inductive predicate, so as to simplify reasoning with it, is not new *per se*, it is for example the case of the technique so-called *step-indexing* [AM01].

Generally speaking, we believe that this case study promotes the idea that a formalization involving complex objects, such as inductive predicates, can be performed within an environment based on automatic theorem proving: it does not necessarily require the use of an interactive proof assistant.

Related work. Our path resolution algorithm is indeed some kind of graph traversal, and its formal proof could be compared with those of standard graph algorithms. In particular, there exists a collection of such graph algorithms proved using Why3 due to Chen and Lévy[5] [CL17]. It seems that the notion of symbolic links adds a significant difficulty when reasoning about graph traversal, which required the use our technique of indexing with height. More recently, the idea of indexing an inductive predicate by a *skeleton* was used by Jeannerod et al. for reasoning, using Why3, about the semantics of Shell scripts [JMT17].

There exists an increasing amount of work on formal reasoning about Unix, file systems and shell scripts, which are too numerous to cite. Though, our work does not pretend to contribute in this category, since our algorithm for path resolution is somehow a theoretical one, that does not need to be considered in practice in operating systems: first the need for a complete algorithm is not important in practice, second our algorithm would be inefficient in practice because it uses extra memory. When formalizing file systems more closely to their practical implementations, the issue of symbolic links is certainly not the main one. For example, Ridge *et al.* in 2015 built a large and detailed formalization of POSIX file system for the purpose of testing real world system [RST+15]. They emphasize the intricacies of path resolution, but do not discuss the possibility of loops: they model the pragmatic approach limiting the number of traversed links. On the contrary they emphasize other issues, for example the fact that path resolution may be performed non-atomically in a concurrent environment. Concurrency is clearly a very difficult issue that we don't take into account in our formalization.

A related work where symbolic links are specifically taken into account is by Ntzik and Gardner in 2015. They propose a framework based on an ad-hoc separation logic to reason about Unix commands that modify the file system [NG15]. Their main goal is to verify the functional behavior of complex commands like `rm -r` that traverse an arbitrary-size part of the file system. In their settings, the question of whether resolution is complete is not considered, and in fact a loop in resolution may induce a non-terminating behavior.

ACKNOWLEDGMENTS

We gratefully thanks N. Jeannerod, R. Treinen and the anonymous reviewers for their useful feedback on this work.

References

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

[5] pauillac.inria.fr/~levy/why3/

[BFMP15] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6):709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[CCM16] Ran Chen, Martin Clochard, and Claude Marché. A formal proof of a Unix path resolution algorithm. Research Report RR-8987, Inria, December 2016.

[CL17] Ran Chen and Jean-Jacques Lévy. A semi-automatic proof of strong connectivity. In *9th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, Heidelberg, Germany, July 2017.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[IEE] IEEE and The Open Group. *POSIX.1-2008/Cor 1-2013*. http://pubs.opengroup.org/onlinepubs/9699919799/.

[JMT17] Nicolas Jeannerod, Claude Marché, and Ralf Treinen. A formally verified interpreter for a shell-like programming language. In Andrei Paskevich and Thomas Wies, editors, *9th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, Lecture Notes in Computer Science, Heidelberg, Germany, July 2017. Springer.

[NG15] Gian Ntzik and Philippa Gardner. Reasoning about the POSIX file system: Local update and global pathnames. In *Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 201–220. ACM, 2015.

[RST+15] Tom Ridge, David Sheets, Thomas Tuerk, Anil Madhavapeddy, Andrea Giugliano, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *25th ACM Symposium on Operating Systems Principles*, 2015.